# Overview of Fortran 90/95

## Names

- starts with a letter
- letters, digits, and underscores ( _ )
- up to 31 characters long

Next slide

# Program Form

```
! No special columns
! Upper and lower case letters
! Case insensitive

x = y + 4  ! Trailing comments

! Continuation on line to be continued
print *, &
      x, y, polar_coordinates

! Multiple statements per line
Temp = a; a = b; b = Temp

! Significant blanks
do i= 1, 99  ! First blank required
```

Previous slide Next slide

Learn more about source form.

# Declarations

A new form of declaration, called an entity-oriented declaration, allows all of the characteristics of one object to be specified in one statement.

Each such declaration begins with the type, followed by other attributes (e.g., `dimension`, `save`, `allocatable`), followed by a double colon and the list of names to have those attributes.

Here are some examples:

```
real :: x, y, polar_coordinates
integer, dimension(10,10), save :: counts = 0
```

Previous slide Next slide

Learn more about data initialization.

# Parameters

A parameter (named constant) may be given its type and value in one statement.

```
integer, parameter :: number_of_states = 50
```

Previous slide Next slide

Learn more about parameters.

# Style of declarations

With these new forms, there are too many ways to write declarations One style should be picked and used consistently.

Here are a few of the equivalent ways to declare the variable NAME to be a 4x5 array of character strings, each of length 20.

```
character NAME (4, 5) * 20
character NAME (4, 5) * (20)
character :: NAME (4, 5) * (20)
character * 20 NAME (4, 5)
character * (20) NAME (4, 5)
character * (20) :: NAME (4, 5)
character (len = 20) :: NAME (4, 5)
character (len = 20), dimension (4, 5) :: NAME
character, dimension (4, 5) :: NAME * 20
character, dimension (4, 5) :: NAME * (20)
```

Previous slide Next slide

Learn more about character data.

# The F Programming Language

As illustrated by the declarations in the two previous slides, there are often too many ways to do things in Fortran. To be consistent and portable and to ensure that you are using modern constructs when writing new code or enhancing older codes, use F. F is a subset of Fortran 90/95 consisting of the moder features; not included are redundant, error prone older ways of doing the same things. [Note: F is not suitable for compiling legacy codes because many Fortran 77 and Fortran 90 features are not included.]

There are F implementations for Windows 95/NT, most Unix workstations, and the Mac. Previous slide
Next slide

# Size of Statements

- Maximum line length is 132 characters.
- Maximum number of continued lines is 39.
- So maximum statement length is 40x132 = 5280 characters.

There is a free source form converter (fixed-to-free). Previous slide Next slide

# Continuation of Character Strings

An ampersand placed on the *continued* line indicates that the source is to start in the next position. This scheme may be used to break in places where white space is not allowed.

```
text = "This is a real&
         &ly long string."
tax = tax_&
      &rate * income
```

Previous slide Next slide

# Rules for Free and Fixed Form

It is possible to write programs in a way that is acceptable as both free source form and fixed source form. The rules are:

1. Put labels in positions 1-5.
2. Put statement bodies in positions 7-72.
3. Begin comments with an exclamation (!) in any position except 6.
4. Indicate all continuations with an ampersand in position 73 of the line to be continued and an ampersand in position 6 of the continuing line.

Previous slide Next slide

# Exercise

1. Write and run a program that computes the sum of the integers 1 through 9, preceded by a short message explaining what the output is.

Previous slide Next slide

# Data Types

There are five intrinsic data types:

- integer
- real
- complex
- logical
- character

The programmer may define additional (derived) data types.

Learn more about the integer type.
Learn more about the real type.
Learn more about the complex type.
Learn more about the logical type.
Learn more about the character type. Previous slide Next slide

# Kind Parameters

Each type has one or more *kind*s that indicate the machine representation of values of that type and kind. They are indicated by integers with system-dependent values.

A variable may be declared to have a particular kind:

```
real (2) :: x, y, z
integer (kind = short) :: q28, much_less
character (len = 20, kind = kanji) :: name
```

Each type has a *default* kind. The real type has at least one other kind that corresponds to Fortran 77 double precision. There must be the same kinds of complex as there are for real. There must be at least one (the default) kind for logical and character. There may be other kinds for any of the data types.

There are conversion functions to convert values to a data type and kind.

Learn more about type conversion intrinsic functions.
Learn more about kind type parameters. Previous slide Next slide

# selected_real_kind

This intrinsic function produces a kind number of a real representation having a certain minimum precision, and a minimum range, if given. The following gives a kind that has at least 9 digits of precision and a range of values to $10^{70}$:

```
selected_real_kind (9, 70)
```

Previous slide Next slide

# selected_int_kind

selected_int_kind permits a range of integers, up to a given number of digits. For example, integers declared to have kind

```
selected_int_kind (6)
```

may have values between -999,999 and +999,999. Previous slide Next slide

# Declaring Variables with Kinds and Writing Nondefault Kind Constants

```
integer, parameter :: &
   single = kind (0.0), &
   double = 8  ! Processor dependent
   . . .
real (kind = 8) x, y, z
real (double) :: d, e, f
real (selected_real_kind (10, 80)) &
      :: p, q, r
   . . .
call s (d)
   . . .
subroutine s (x)
real (8) x
real (kind = kind (x)) :: t1, t2
t1 = 1.0_8 / 3
   . . .
```

Previous slide Next slide

```
integer (kind = 5) j1, j2
integer (selected_int_kind (6)) j3

j1 = 49_5 ** 2
j3 = 999999
```

Previous slide Next slide

```
integer, parameter :: greek = 6
   . . .
character (len = 5, kind = greek) &
   greek_island
   . . .
greek_island = greek_'μμμμμ'
```

Note that the kind *precedes* the rest of the character constant. Previous slide Next slide

# `implicit none`

The `implicit none` statement turns off all implicit typing. It should be used in every new program.

Other uses are discouraged, but click here if you need to know how it works.

Learn more about implicit typing. Previous slide Next slide

# Precision-Related Intrinsic Functions

```
digits
epsilon
exponent
fraction
huge
maxexponent
minexponent
nearest
precision
radix
range
rrspacing
scale
set_exponent
spacing
tiny
```

Learn more about inquiry and model intrinsic functions. Previous slide Next slide

# Exercise

1. Determine the number of one real kind that has precision greater than that of the default real kind on your computer system.

Previous slide Next slide

# Expressions

Expressions are the formulas of FORmula TRANslation.

They may appear in many different contexts in a Fortran program, but the rules are essentially the same as in Fortran 77.

There are two special forms of expressions:

- specification expressions
- initialization expressions

The rules about what may appear in these special expressions are quite complicated. The end result is that the value of an initialization expression (e.g., the value of a parameter) is known at compile time and the value of a specification expression (e.g., array bounds) is known when a procedure is entered. Follow the links above to learn more.

Learn more about expressions. Previous slide Next slide

# Operator Precedence

The following table reviews the precedence of Fortran operators, including user-defined operators.

| Category of operator | Operator | Precedence | In context of equal precedence |
|---|---|---|---|
| **Extension** | Unary defined operator | Highest | N/A |
| **Numeric** | `**` | . | Right-to-left |
| | `*` or `/` | . | Left-to-right |
| | Unary `+` or `-` | . | N/A |
| | Binary `+` or `-` | . | Left-to-right |
| **Character** | `//` | . | Left-to-right |
| **Relational** | `.eq.`, `.ne.`, `.lt.`, `.le.`, `.gt.`, `.ge.` | . | N/A |
| | ==, /=, <, <=, >, >= | . | N/A |
| **Logical** | `.not.` | . | N/A |
| | `.and.` | . | Left-to-right |
| | `.or.` | . | Left-to-right |
| | `.eqv.`, `.neqv.` | . | Left-to-right |
| **Extension** | Binary defined operator | Lowest | Left-to-right |

Previous slide Next slide

# Formation, Interpretation, and Evaluation of Expressions

The syntax rules indicate how to form valid expressions (syntax); other rules, such as operator precedence, indicate how to interpret an expression (the semantics). Once all this is established, the system may actually do the computation in a completely different order, as long as the method is *mathematically equivalent* and groupings by parentheses are not modified. Previous slide Next slide

Examples of equivalent expressions:

| Expression | Equivalent |
|---|---|
| `x + y` | `y + x` |
| `x * y` | `y * x` |
| `- x + y` | `y - x` |
| `x + y + z` | `x + (y + z)` |
| `x - y + z` | `x - (y - z)` |
| `x * a / z` | `x * (a / z)` |
| `x * y - x * z` | `x * (y - z)` |
| `a / b / c` | `a / (b * c)` |
| `a / 5.0` | `0.2 * a` |

Previous slide Next slide

Examples of expressions that are not equivalent:

| Expression | Not equivalent |
|---|---|
| `i / 2` | `0.5 * i` |
| `x * i / j` | `x * (i / j)` |
| `i / j / a` | `i / (j * a)` |

Previous slide Next slide

Examples of equivalent expressions that may not be evaluated because of parentheses:

| Expression | Not transformable to: |
|---|---|
| `(x + y) + z` | `x + (y + z)` |
| `(x * y) - (x * z)` | `x * (y - z)` |
| `x * (y - z)` | `x * y - x * z` |

Previous slide Next slide

## Exercise

1. Write a program to compute the quantity $e^{i\,pi}$. The constant *pi* can be computed by the formula *pi* = 4 * atan (1.0) since tan (*pi*/4) = 1. The complex constant *i* can be written (0,1). The built-in function exp (z) is used for raising the mathematical constant *e* to a power. The output should look like:

```
run eipi

 The value of e to the power i*pi is _ _ _
```

Previous slide